David Kelly, dmk257
Ross Hanson, reh237
CS 5413: Final Project

NetSlice Bottleneck

Abstract:

This paper provides a basic analysis of the performance of NetSlice versus two similar frameworks, Intel DPDK and netmap. NetSlice, while extremely portable and scalable up to 30Gbps, hits a bottleneck when attempting to scale it further. Intel DPDK and netmap, on the other hand, are capable of scaling up to 40Gbps. We hope to find this bottleneck and provide evidence that NetSlice can be also be scaled up to 40Gbps.

Introduction:

High performance packet processing libraries offer developers the ability to leverage emerging hardware in order to develop their applications beyond just using the raw socket. This allows developers to create high speed networking applications on commodity hardware - something that is extremely useful in the cloud. Because of its usefulness, several libraries have been created to give developers this control. Three of these include:

- Intel DPDK - Developed by Intel for Intel hardware. It provides the performance necessary for high speed packet processing, but is limited in its portability.
- netmap - Developed to provide the same performance as other packet processing libraries but be portable across Unix machines. It's portability is limited, though, by the fact that special drivers must be used in order to allow its zero-copy of packets between kernel and user space.
- NetSlice - Developed to provide the same performance as above, but with true portability. NetSlice does minimal copying between kernel and user space and requires no special drivers to run.

Intel DPDK and netmap fall short on their ability to build applications that are truly portable across all Unix machines. This is where the NetSlice comes in. By using minimal copies (and thus no special drivers), NetSlice would allow for a truly portable application to be developed. Despite this goal, though, NetSlice has one problem: currently it appears to be bottlenecked at 30Gbps, while both netmap and Intel DPDK seem to be able to scale up to 40Gbps. It is our goal to reproduce the results of this bottleneck, analyze NetSlice's performance to determine its cause, and provide evidence that NetSlice can in fact be scaled to 40Gbps.

Architecture:

To test the system, we interconnected three fracus machines: one to act as a packet source, one to act as a middlebox, and one to act as a packet sink. The source will generate packets up to 40Gbps (10Gbps per line) and send them towards the middlebox. The middlebox will have one of the three packet processing libraries loaded and act as a simple

bridge, forwarding packets onto the sink. The sink will finally receive the packets, and report how many it has seen. This will allow us to determine how many packets were dropped between the source and the sink, and thus determine how much traffic the individual libraries loaded on the middlebox can handle.

Compute24, compute25, and comput26 were used in the Fractus cluster, acting as the source, middlebox, and sink, respectively. The machines and interfaces were interconnected according to the diagrams in Figure 1 and Figure 2.

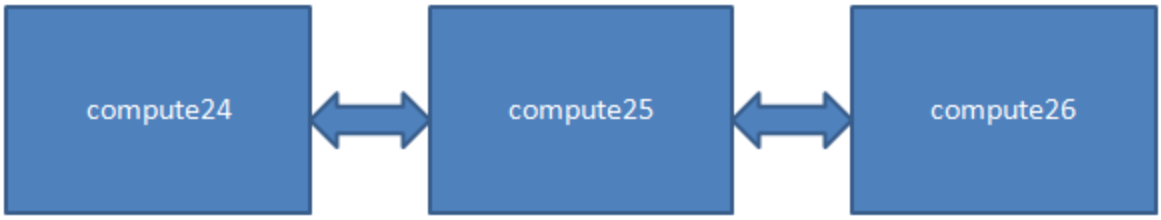

Figure 1: A high level diagram of how the fractus machines were connected.

24:eth6 -> 25:eth7	25: eth6 -> 26:eth6
24:eth7 -> eth9	25: eth8-> 26:eth7
24:eth8 -> 25:eth11	25: eth10-> 26:eth8
24:eth9 -> 25:eth13	25: eth12 ->26:eth9

Figure 2: The interconnected interfaces between compute24, compute25, and compute26. For example, 24:eth6 -> 25:eth7 indicates that eth6 on compute24 was connected to eth7 on compute25.

Analysis:

Reproducing bottleneck results:

The first step in our analysis was to reproduce the results of the bottleneck seen previously on NetSlice. To do this, we loaded the middlebox with either netmap, NetSlice, or Intel DPDK and then used the source to generate packets towards it, with the middlebox forwarding the results to the sink and the sink reporting how many packets got through. The results can be summarized the in Figure 3.

	netmap	NetSlice	Intel DPDK
1 Line	9.91	9.2617	9.91
2 Line	19.76	17.6279	19.76

3 Line	29.68	27.9451	29.33638814
4 Line	*	31.8142	*

Figure 3: The results of running a bridge on the middle box and passing packets over it.

Note: netmap does not contain a 4th line due to an error where only 3 netmap instances could be generated at once. Netmap was also used to send packets to Intel DPDK due to problems receiving packets from netperf with Intel DPDK. This meant that we were unable to send 4 lines worth of packets towards DPDK and our results were also bounded by how quickly netmap could generate packets.

As can be seem from the table, there is a particularly stark drop off in NetSlice's performance at 40Gbps. Up to 30Gbps, NetSlice has comparable performance to netmap and Intel DPDK, if only just slightly slower. At 40Gbps, though, NetSlice can only handle 31.8142Gbps.

Determining the cause of the bottleneck:

In order to determine the cause of the bottleneck, three methods of analysis were used:

- "Static analysis": Due to the size of NetSlice, it was somewhat feasible to read through the code and look for problems such as algorithmic deficiencies, blatant problems in the code, and in general get a better idea of where there might be problems. While it was no glaring issues were immediately clear, we were able to get a very rough idea on where the bottleneck may occur due to the nature of several functions. We identified unlocked_netslice_ioctl(), netslice_readv(), netslice_writev(), netslice_read_default(), netslice_write_default(), and netslice_poll() as being possible causes of the bottleneck, as all of these functions are run more than once and each have some form of expensive operations that take place.
- Fuzz Testing: We varied a number of parameters throughout the NetSlice codebase and recorded changes in the observed throughput, both with and without profiling. Some changes, such as modifying schedule() to schedule_timeout() made slight differences in profiling but no observable changes to the overall throughput. Others, such as the use of temporary accumulator variables made large differences in profiling as well as changes to the observed throughput.
- OProfile: OProfile provided the best results for us in finding the bottleneck and allowed us to identify the main bottleneck. OProfile is a performance profiler that allows the user to not only profile user space applications, but also kernel modules and the kernel itself. Once profiled, it can then annotate the source code with a series of metrics that correspond to specific lines of code (for example, how many cycle samples did a certain line encounter, or how many L2 cache misses were there for a certain line). We relied on the cycle samples metric to determine how much time the machine was spending at each section of code (a cycle sample is defined as one sample per a

certain number of cycles). A detailed set of instructions on how to use OProfile are given in the appendix of this paper.

Analysis Results:

After profiling the kernel module and annotating the source code, we were able to determine that the netslice_readv() function took around five times longer to run than the next closest function, netslice_writev(). In one of our tests, 84,432 cycle samples were spent within netslice_readv(), while only 18,180 were spent netslice_writev(). Of the samples within netslice_readv(), roughly 33,000 of them were spent in a synchronization loop which transferred received skbs from a waiting queue to a "to-be-returned" queue. The code for this loop can be seen below. Another thousand were spent accessing and modifying the iov array. Surprisingly, out of the number of samples in this function, relatively few of them were spent copying data between kernel and userspace (on the order of several thousand, but significantly less than the number of samples during the iov code).

```
spin_lock_bh(&rcv_queue->lock);
list_for_each(skb_item->list, &rcv_queue->skbs)
{
        /* can return at most count items */
  if (fetched_cnt >= count)
                break;

        /* unchain the element */
        list_del(&skb_item->list);
        rcv_queue->bytes_len -= skb_item->skb->len;
        recv_len_diff++;


        /* chain to the return list */
        list_add_tail(&skb_item->list, &ret_skbs);
        fetched_cnt++;
}
spin_unlock_bh(&rcv_queue->lock);
```

Figure 4: This loop was responsible for about 30% of the samples in profiling

Optimizations:

Three main optimizations were attempted:

- Compile time optimizations: Upon our initial "static analysis" of NetSlice, we noticed that it was compiled with 02 rather than 03. As 03 provided further optimizations (such as attempting to include vectorized instructions in loops) and thus further performance gains, we compiled with 03 to see the effect.
- schedule() vs. schedule_timeout(): schedule() causes a process to be removed from the run queue and informs the scheduler than another process can be run instead of it.

The problem with this is that it provides no guarantee that the process will be woken up anytime soon. On a particularly busy system, this can cause a problem as other processes may not yield execution in a timely manner. Due to how cpu intensive NetSlice is when running up to 40Gbps, we believed this issue may have been occurring. A solution to this was to use schedule_timeout() which takes in a delay in jiffies (one jiffy is equal to the time between interrupts, in many Unix machines this is roughly 10ms). After the delay, the process will begin running again. Alternatively, if there is a signal, the process can begin running sooner than the delay. A detailed description of schedule() and schedule_timeout() can be found here: http://www.makelinux.net/ldd3/chp-7-sect-3 .

- Batching expensive writes: From the OProfile data, we were able to determine that a synchronization loop within netslice_readv() was the most likely cause for the bottleneck. Specifically, within that loop one line of code where a pointer passed into the function was modified was responsible for most of the samples attributed to that loop. After digging into the code, we learned that this pointer was passed into the function by the linux kernel, and thus may possibly be rather expensive to write to due to something going on behind the scenes with that value. A temporary variable was created to back the updates to, and then after the loop the updates were written to the pointer at once.

Of these optimizations, only the batching optimization had any significant performance increases, and those were minimal at best. Both the compile time optimization and switching to schedule_timeout() seemed to make no difference. For the schedule() optimization, we believe this made no difference due to NetSlice being the only cpu intensive application being run on the machine and, even though it was very intensive, because of how often it called schedule(), the scheduler had many opportunities to switch between processes and not keep others waiting too long.

As for the batching operation, we were able to see minimal successes described in Figure 5. In the 4 line case, we see about a 400Mbps increase in its bandwidth compared to before the optimization. It is worth noting that several batch optimizations were made within that four loop. As each successive optimization was made and tested, the profiler would indicate that another line within the synchronization loop would be taking up the majority of the samples. Upon fixing one, another would appear. While the batch operations appear to have helped, it is possible that these large focuses of samples on one line may be due to a profiler mistake, and thus these individual lines were not the main cause of the bottleneck, but the synchronization loop as a whole. This would explain why the batch optimizations had a some effect, but not as large as expected based on the number of samples recorded for those lines.

It is also worth noting the effect that the profiling overhead has on NetSlice. During its normal operation without optimizations, NetSlice would be able to handle roughly ~31Gbps. During profiling, the extra overhead would cause it to handle ~7Gbps instead. This could mean that the act of profiling NetSlice may have affected the results from profiling. This can further be seen in the results after the batch optimizations. Afterwards, NetSlice was roughly

able to handle ~32Gbps, while during profiling it was able to handle ~12Gbps. While the batch optimizations have helped, they may have helped more in speeding up NetSlice while it was being profiled than otherwise. For this reason, the results from the profiler should be used as a guide to find the bottleneck, not necessary to find the exact cause of it.

	NetSlice
1 Line	9.195
2 Line	17.922
3 Line	27.926
4 Line	32.204

Figure 5: The effects of the batch optimization on NetSlice

Related Work:

Intel DPDK and netmap are the most relevant works to this research. Each group has taken a different approach to providing a library for high performance packet processing, and both groups are separate from what NetSlice is attempting. While separate works, their successes can provide useful ideas and starting points for NetSlice as well as giving something to improve on top of.

Future Work:

For future work, it would be worthwhile looking into several areas:

- Using OProfile to determine L2 cache misses: Given fractus' hardware, one of the events that OProfile can listen to rather that cycle samples is L2 cache misses. Using this metric, we may be able to getter a better idea of where exactly the bottleneck is being caused and if L2 cache misses play a role in it.
- Identifying and fixing the bottleneck: This goal is mostly a given. Using the data here, we have provided a stepping stone for the next researchers to determine exactly what is causing the bottleneck. Once it has been identified, a fix can be made in order to further increase the performance of NetSlice.
- Building a full fledged test application: Several small applications already exist from previous developers of NetSlice, but it would be interesting to see how NetSlice performs under heavy user loads and complicated applications.

Conclusions:

From the data gathered from analyzing NetSlice, we can conclude that the bottleneck most likely comes from the synchronization loop within the netslice_readv() function. As the main difference between NetSlice and netmap is netmap's zero-copy ability, we believe that given work on this area of code, it is possible that NetSlice can in the future be scaled to the

performance of netmap and Intel DPDK while still keeping its promise to be portable across most Unix operating systems.

Appendix:

Bugs encountered:

- NetSlice CPU bug: This was easily the largest hurdle we encountered. During the course of use, we would observe our middle machine, compute25, gradually slow down without any visible changes in monitoring programs such as htop. Eventually, malloc calls appeared to begin failing, resulting in programs such as ssh and oprofile failing, eventually rendering the system unusable. Upon restart, there would be a cryptic message in the dmesg logs indicating a CPU error in the netslice module, under the "general protection fault" category. We were unfortunately unable to diagnose the exact cause of this bug but did notice that it appeared more frequently as we unloaded/loaded the module more, instead of being relative to the number of packets processed.
- netmap unable to run more than 3 instances: Unmodified netmap was unable to run more than 3 instances on our machines at once. Starting instances 1-3 would proceed as expected, however the launch of the fourth would return immediately with an error about about being unable to bind to the given interface. If one of the earlier instances was killed, this command would then proceed as expected. Ki speculated that this had to do with contention within the netmap module, though we were unable to verify this.

A brief tutorial on running OProfile:

NOTE: This tutorial uses OProfile 0.9.9. OProfile 1.0.0 is the most recent version as of the writing of this paper, but this version deprecated the opcontrol command in favor of operf. While this is not necessarily a bad thing, unfortunately the documentation on using operf with a kernel module is rather lacking at the moment. For this reason, this tutorial will instead use OProfile 0.9.9 and opcontrol.

NOTE: Portions of this tutorial are taken from http://www.lainoox.com/profiling-kernel-modules-using-oprofile/ . This tutorial is useful, but unfortunately makes several mistakes and does not show how to correctly annotate kernel modules. This tutorial http://www.ibm.com/developerworks/systems/library/es-oprofile/ shows how to correctly annotate source code, but does not show how to do this for a kernel module. Below are the full, combined steps that worked for us.

1. First, the kernel must be recompiled in order to add the necessary flags such that OProfile can profile it.
    a. An easy way to get the kernel source is through git. Choose the correct kernel for your machine, rather than the example used here.
        i. git clone git://kernel.ubuntu.com/ubuntu/ubuntu-precise.git
        ii. cd ubuntu-precise
        iii. git tag -l
    b. Checkout the correct branch for your kernel

i. git checkout -b mybranch Ubuntu-lts-3.8.0-34.49_precise1

c. Recompile the Kernel. These instructions worked best for me: http://mitchtech.net/compile-linux-kernel-on-ubuntu-12-04-lts-detailed/ . When you use the "make menuconfig" command, make sure the following properties are set:
   i. General setup -> Profiling support = y
   ii. General setup -> OProfile system profiling = y
   iii. Kernel hacking -> Strip assembler-generated symbols during link = n
   iv. Kernel hacking -> Compile the kernel with debug info = y
   v. Networking support -> Networking options -> 802.1d Ethernet Bridging = m

2. Install OProfile 0.9.9 if you do not already have it. It can be found here: http://oprofile.sourceforge.net/news/
3. To begin profiling, run the following commands
   a. Compile netslice_mod.ko with the -g option to allow for debugging information
   b. sudo opcontrol --init #Initializes the profiling session
   c. sudo opcontrol --vmlinux=/path/to/recompiled/kernel/vmlinux #Set the kernel to be profiled
   d. sudo opcontrol --start #begins profiling the kernel and all modules
   e. Run your code that you want profiled.
   f. sudo opcontrol --dump #dumps the data gathered from profiling to disk
   g. sudo opcontrol --deinit #ends the profiling session
   h. sudo opannotate --source --output-dir /absolutePath/ToADirectory/ForTheAnnotatedSourceFiles --image-path /absolutePath/ToTheDirector/Where_netslice_mod.ko/wasCompiledInto #Annotate the source files based on the the profiling data, store the annotated files in output-dir directory, and point the annotation command to the directory which contains netslice_mod.ko so that it can also be annotated.
4. Now, the annotated files with the default metric (cycle samples, most likely) will be stored in the folder specified during opannotate.

A brief tutorial on running NetSlice at line rate:

1. To begin, you will need to set the irq affinitiies to ensure that packets from the NICS are processed on the appropriate CPU's. This can be done using the provided "set-irq-affinity" script on each of your target interfaces for each machine. For instance, to run this on eth6 through eth10, use a command like the following:

   "for ((x=6;x<10;x++)); do ./set-irq-affinity.sh eth${x} 1 0; done"
2. You will also need to enable rss optimizations on your receiving machines. Ki Suh has been generous enough to supply a script which does this, called "enable_rss.sh." This should be run on your middle and sink machines.

3. Next, you'll need to build and load the NetSlice module on your middle machine. This also requires you create the appropriate number of slice special files in a common location (such as /dev/netslice), which can be done using the "create_dev.sh" script.
4. Ensure a netserver is running on your sink machine
5. Run at least 1 netperf instance per NIC on your source machine, ensuring that each instance is bound to the appropriate interace via the "-H" option. Upon completetion, these commands should indicate the bandwidth available using NetSlice.

Interim Report 2:

1. One paragraph on the high level context/motivation for what you are proposing.

   From an education standpoint, we would like to learn about the various packet processing frameworks on a lower level and their performance at bleeding edge speeds. From a practical perspective, we would like to further the work Cornell has done on the NetSlice library. NetSlice's increased portability compared to netmap and DPDK is a desirable feature in a "write once, run anywhere" world; ideally, this portability should not come with a performance penalty.

2. A paragraph (or bullet points) on what you will do to carry out the project: e.g. "We will get access to the NSF GENI Testbed with a SoNIC slice and implement our own SoNIC-enabled network measurement architecture for monitoring available bandwidth" etc. You should also accompany these actions with an estimated date of completion (time schedule for the project). You can evolve this plan later if needed; the one you file is an initial concept.

   - We will get access to three hosts from the Fractus cluster (ETA: 9/26)
     - Each host must be capable of sending and receiving 40Gb/s
     - Host one must be wired to host two and host two must be wired to host three so that traffic traffic may be generated at host one and sent through host two to host three
   - Software must be developed/installed for host one such that it can generate and send traffic at 40Gb/s to host two (ETA: 10/3)
     - Traffic will likely be sent as UDP packets at first to avoid the added complexity of the TCP handshake and congestion algorithms
     - Once UDP traffic generation is implemented, TCP traffic generation will be implemented
     - We are considering using Pktgen from Wind River for packet generation in order to save development time: https://github.com/Pktgen/Pktgen-DPDK
   - Software must be developed/installed for host three such that it can receive traffic at 40Gb/s (ETA: 10/3)
   - DPDK, NetSlice, and netmap must be implemented on host two so that the host can receive traffic from host one and forward it to host three (ETA: 10/17)
   - Develop simple utility to switch between packet processing frameworks (ETA: 10/17)
   - Implement metrics on host one, two, and three in order to get details on how each host is processing (10/17)
   - NetSlice must be analyzed in order to determine the bottleneck that is stopping it from achieving 40Gb/s (ETA: 11/7)
   - Implement a fix for the bottleneck in order to achieve 40Gb/s (ETA: 11/21)

- Implement a test application (such as a TCP accelerator) using the three libraries to show them in use (ETA: 12/12)

3. A paragraph (or bullet points again) explaining how you will demonstrate the project (on completion, we will have a visual demo and a poster. The demo will show....) This can evolve over time too.

    - On completion, we will have a poster which summarizes our design and implementation
    - Using the metrics created above, a live demo can be set up where we forward 40Gb/s of traffic through host two and see the effect that each of the libraries has on performance
    - Similarly, we can use the same setup to demo the test application in real time

4. Of the team of three, who will do what? How often will you meet? How many hours per week will you work on the effort? How many credits are you taking: can you really spend that number of hours per week? We allow teams of three for any project.

    - The development will be split up into multiple phases. In each phase, the work will be split evenly among the two developers, with each developer picking up a new task when the previous is finished:
        - In the first phase, Fractus will be set up and packet generation solutions will be looked into.
        - In phase two, DPDK, NetSlice, and NetMap will be installed on machines, a utility or script will be written to switch between them, and metrics will be developed to gather data on how the systems are functioning.
        - Phase three - likely the longest phase - will be spent digging into and analyzing NetSlice to find and fix bottlenecks in the code.
        - in the fourth and final phase, a test application will be implemented in order to display the results of the speedup in a practical manner.
    - Generally speaking, we often spend time working together as we both generally work in the M.Eng lab and other study areas around campus. for that reason, most of our meetings will be ad hoc throughout the week. At a minimum, we will schedule at least one hour per week to discuss the plans for that week.
    - We aim for 10 hours per week per person
    - David Kelly is taking 15 credits this semester. Ross Hanson is taking 19 credits this semester.

Interim Report 2 Status:
As of 10/29/14, here is our current status:

- [ ]We will get access to three hosts from the Fractus cluster (ETA: 9/26)

- [ ]Software must be developed/installed for host one such that it can generate and send traffic at 40Gb/s to host two (ETA: 10/3)
    - Currently using a NetMap, but also have the capabilities to use Wind Rivers pktgen which is based on Intel DPDK
- [ ]Software must be developed/installed for host three such that it can receive traffic at 40Gb/s (ETA: 10/3)
    - We can use the same software as above to sink and source packets.
- [ ]DPDK, NetSlice, and netmap must be implemented on host two so that the host can receive traffic from host one and forward it to host three (Original ETA: 10/17; new ETA: 11/3)
    - All three frameworks are installed, however we have not implemented the appropriate port juggling for NetSlice to spread evenly amongst all cores
- [ ]Develop simple utility to switch between packet processing frameworks (Original ETA: 10/17; new ETA: 11/3)
    - We have created install and uninstall scripts for DPDK. netmap is simple enough that we have not had to create scripts yet. We have combined some of the existing scripts to quickly deploy and test our NetSlice implementation.
- [ ]Implement metrics on host one, two, and three in order to get details on how each host is processing (Original ETA: 10/17; new ETA: 12/5)
    - We have been able to gather basic metrics on netmap and DPDK. We have been having issues getting netmap to work on 4 interfaces, and thus do not have a metrics for that. We have also transitioned from netmap's pkt-gen utility to iperf and netperf for load testing.
    - We have been having been having issues gathering data with NetSlice, as it fairly reliably causes a CPU bug which takes down the host. This is described in the next section.
    - Below are some of our current metrics using one core per interface in Gb/s:

	Netmap	NetSlice	Intel DPDK
1 Line	9.91	~9	9.91
2 Line	19.76		19.76
3 Line	29.68		29.33638814
4 Line			

- [ ]NetSlice must be analyzed in order to determine the bottleneck that is stopping it from achieving 40Gb/s (ETA: 11/7, Updated ETA: 12/5)
    - We have leveraged oprofile in order to aid in determining the cause of the bottleneck.

    - In debugging NetSlice, we have gained a higher level of familiarity with the relatively manageable codebase. In doing so, we have discovered an occasional CPU bug that appears to stem from an inappropriately initialized NetSlice filter, though we've set to find the exact cause of this in code. This bug often times leads to a host going down and has slowed our progress in analyzing NetSlice.
- [ ]Implement a fix for the bottleneck in order to achieve 40Gb/s (Original ETA: 11/21, Updated: 12/12/)
    - We have yet to identify the bottleneck.
- [ ]Implement a test application (such as a TCP accelerator) using the three libraries to show them in use (ETA: 12/12)
    - We are planning to leverage a semi-fullly featured firewall that Ross is developing for CS5434 as our sample application. This firewall is being developed in parallel (though loosely coupled) with our NetSlice work.